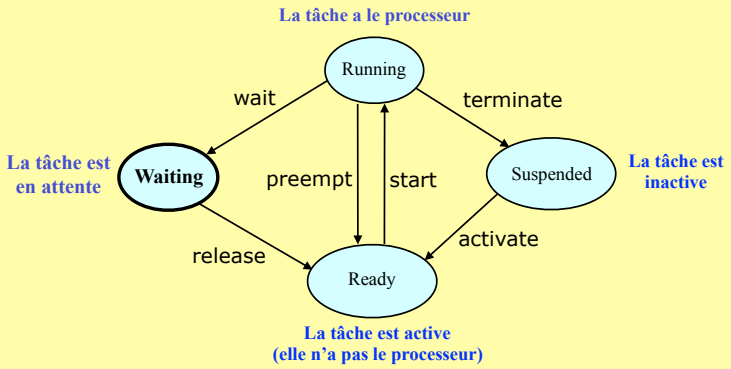
Chapitre 4 –

Autres services de l'exécutif

☞ Les tâches étendues

- une tâche étendue se compose de un ou plusieurs modules de code séquentiel, sans appel système bloquant ;
- via un appel de service (WaitEvent) une tâche étendue se met **en attente de la réalisation d'une condition** ;
- cette condition se rapporte à un « **événement** ». « **L'occurrence** » de l'événement débloque la tâche et la réveille.
- son diagramme d'états a 4 états : l'état « **Waiting** » en plus.

Diagramme des états



Une tâche étendue peut prendre de multiples formes. Voici quelques exemples qui demandent une analyse approfondie pour en saisir le sens :

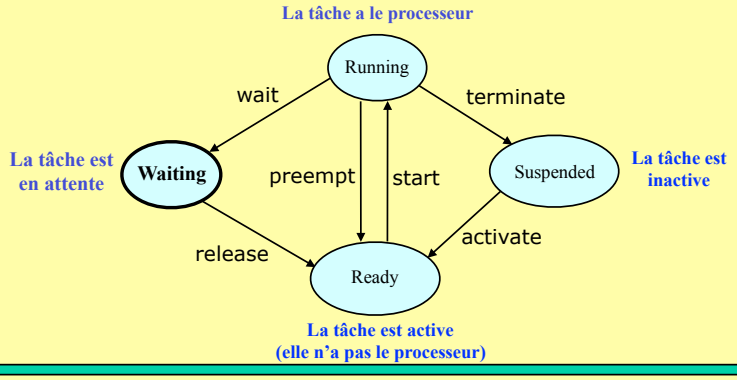
➡ boucle infinie, un module séquentiel déclenché par une occurrence :
while (1) { WaitEvent(...) ; - code - ; }

➡ boucle infinie, deux modules séquentiels séparés par deux synchronisations :
while (1) { WaitEvent(...) ; - code - ; WaitEvent(...) ; - code - ; }

➡ et pourquoi pas une activation par un ActivateTask, même si ce n'est pas l'esprit :
- code - ; WaitEvent (...) ; - code - ; TerminateTask ( ) ;

Exemple d'événement : un capteur à seuil qui détecte le dépassement.
D'une manière générale, toute condition sur l'environnement ou dans le programme, qui devient vraie, engendre une occurrence de l'événement.

Diagramme des états



☞ Le concept d'« événement »

➡ un événement c'est comme un drapeau qu'on lève pour signaler quelque chose qui vient de se produire.

➡ événements **privés** :
1 événement est la propriété d'une tâche étendue
➡ seule la tâche propriétaire peut invoquer le service d'attente

Parler des événements « privés », c'est dire que l'on peut avoir des événements publics. C'est le cas de certains exécutifs pour lesquels l'événement n'est la propriété de personne. Tout le monde peut se mettre en attente sur une occurrence (file d'attente indispensable).

➡ modèle **n producteurs** / **1 consommateur**
➡ - n tâches peuvent invoquer le service de signalisation
(tâches basiques, tâches étendues, ISR)
- 1 tâche (la propriétaire) invoque le service d'attente

Le nombre maximum d'événements d'un tâche est fixé par l'implémentation (16 pour Trampoline).

☞ Les services pour les tâches étendues

Le service WaitEvent

Forme C : Status **WaitEvent** (EventMask Event)
le compte-rendu d'exécution n'est pas utilisé

C'est forcément la tâche en cours d'exécution (donc la tâche propriétaire de l'événement) qui appelle ce service.
Le paramètre est un « **masque** » d'événements :
ensemble d'événements réunis par une **fonction logique OU**.
Si l'événement **n'est pas arrivé**, la tâche **est bloquée**. Elle passe alors dans l'état « Waiting ».
Si l'événement **est arrivé** (s'est produit), la tâche **n'est pas bloquée** et elle continue son exécution;

Remarque : **une ISR n'est pas autorisée à appeler ce service.**

Sur WaitEvent, il suffit d'une occurrence pour sortir de l'attente. Il n'existe aucun blocage si une occurrence au moins est déjà survenue avant l'invocation du service. C'est la tâche qui doit déterminer l'origine de la fin d'attente. Pour cela, elle utilise le service GetEvent qui permet de connaître les occurrences arrivées.

☞ Les services pour les tâches étendues

Le service SetEvent

Forme C :
Status **SetEvent** (Task_Id TaskName, EventMask Event)
le compte-rendu d'exécution n'est pas utilisé

Les paramètres sont :
- le **nom de la tâche** à laquelle appartient l'événement
- un « **masque** » d'événements : tous les événements nommés dans le masque sont déclarés « Arrivé ».

Si la tâche désignée est en attente d'un des événements déclarés « arrivé », la tâche passe dans l'état « Ready ». Si elle est la plus prioritaire des tâches prêtes elle gagnera le processeur.
Si la tâche désignée n'est pas en attente d'un des événements déclarés « arrivé », la tâche qui appelle SetEvent continue.

Remarque : **ce service peut être invoqué par une tâche basique, une tâche étendue ou une ISR.**

☞ Les services pour les tâches étendues

Le service GetEvent

Forme C :

Status **GetEvent** (Task_Id TaskName, EventMask* Event)

le compte-rendu d'exécution n'est pas utilisé

Les paramètres sont :

- le **nom de la tâche** à laquelle appartient l'événement
- l'adresse de la variable qui recevra le masque d'événements

La tâche invoquant ce service continue son exécution.

C'est le service qui permet de savoir quel est l'état des événements de la tâche désignée.
Remarque : n'importe qui peut donc se renseigner sur l'état des événements d'une tâche étendue.

Remarque : **ce service peut être invoqué par une tâche basique, une tâche étendue ou une ISR.**

☞ Les services pour les tâches étendues

Le service ClearEvent

Forme C :

Status **ClearEvent** (EventMask Event)

le compte-rendu d'exécution n'est pas utilisé

Les événements désignés dans le masque d'événement sont placés dans l'état « Non_Arrivé ».

La tâche invoquant ce service continue son exécution.

Dans le cas d'OSEK, puisque l'événement est binaire, on masque toute occurrence d'événement entre le moment où WaitEvent devient passant et le moment où la tâche invoque ClearEvent.
Ce n'est pas le cas avec l'autre modèle.

Remarque : **ce service ne peut être invoqué que par la tâche propriétaire de l'événement.**

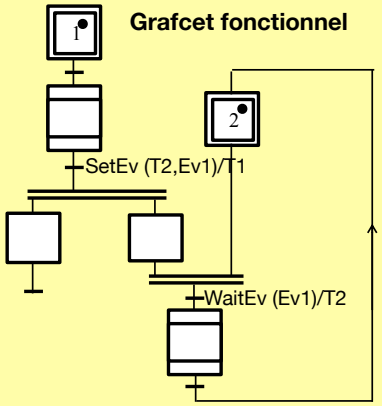
☞ Exemple 1

Synchronisation entre une tâche basique et une tâche étendue.
L'activation de la tâche basique n'est pas montrée.

```
TASK (Tache_1)
{
   - - - ;
   SetEvent (Tache_2, EV1) ;
   TerminateTask ( ) ;
}
```

```
TASK (Tache_2)
{
   while (1)  {
      WaitEvent (EV1) ;
      ClearEvent (EV1) ;
      - - - ;
   }
}
```

Grafcet fonctionnel



Description architecture

```
- - - ;
Event EV1 {
     MASK = AUTO ;
};
- - - ;
```

```
TASK Tache_2 {
     - - - ;
     EVENT = EV1;
     - - - ;
};
```

Le choix AUTO pour l'attribut MASK fait que le compilateur GOIL calcule automatiquement l'affectation des événements sur les bits du mot des événements de la tâche.

La description de l'architecture en langage OIL montre comment une tâche déclare qu'elle possède un événement. Ce dernier est par ailleurs déclaré comme un élément. Si on ne veut pas utiliser l'attribut AUTO, il faut définir, via le masque, la position de l'événement dans le mot. C'est une source d'erreur inutile. Il est préférable de laisser le compilateur engendrer les masques.

☞ Exemple 2

Deux événements, l'un signalé par une tâche basique, l'autre signalé par une ISR2.

```
ISR (Gestion_IT_CC8IO)
{
   - - - ;
   SetEvent (Tache_2, EV2) ;
   TerminateISR2 ( ) ;
}
```

```
TASK (Tache_1)
{
   - - - ;
   SetEvent (Tache_2, EV1) ;
   TerminateTask ( ) ;
}
```

```
TASK (Tache_2)
{
   int Masque;

   while (1)  {
      WaitEvent (EV1 | EV2) ;
      GetEvent (Tache_2, &Masque) ;
      if (Masque & EV1)  {
            - - - ;    // c'est EV1
      } else  {
            - - - ;    // c'est EV2
      }
      - - - ;
   }
}
```

Description architecture

```
- - - ;
Event EV1 {
     MASK = AUTO ;
};
Event EV2 {
     MASK = AUTO ;
};
TASK Tache_2 {
     - - - ;
     EVENT = EV1 ;
     EVENT = EV2 ;
     - - - ;
};
```

☞ Notion de ressource critique

➡ Une **ressource** (resource en anglais) est quelque chose qui peut être **partagée** (utilisée) par plusieurs utilisateurs.

- par exemple : des données en mémoire, afficheurs LCD, une imprimante …

➡ L'accès à une ressource partagée par plusieurs utilisateurs concurrents doit se faire avec précaution, sous peine de corruption des données.
Par exemple : un fichier peut être lu par plusieurs utilisateurs concurrents simultanés, mais l'accès en écriture doit être exclusif.

➡ Une ressource est dite **en exclusion mutuelle** lorsqu'**un seul accès** est autorisé à un instant donné. Elle est alors qualifiée de **critique**.

- par exemple : utilisation exclusive par une tâche de données en mémoire, réservation d'un afficheur LCD pour usage temporaire exclusif …

➡ L'accès en exclusion mutuelle à une ressource peut se faire :

- soit en étant en **ordonnancement non préemptif**
- soit en respectant un **protocole particulier** à l'aide de services de l'OS

☞ Notion de ressource critique

On peut donner un exemple avec une simple variable dont l'accès ne peut pas être atomique.
Exemple :

Tache_1 : - - - ; V1 = V1 + 2 ; - - - ;
Tache_2 : - - - ; V1 = V1 + 5 ; - - - ;

Pour réaliser cela en assembleur, il va falloir faire une lecture de V1 puis l'opération, puis une écriture.

On peut envisager la situation où avec V1 initial = 5 :

Tache_1 fait la lecture (V1 lue = 5), puis est préemptée par Tache_2 qui fait tout puis se termine (V1 écrit = 10), Tache_1 terminant alors son écriture (V1 écrit = 7). La valeur n'est pas ce qu'elle aurait dû être (12) si les deux opération d'accès avaient été sérialisées (peu importe l'ordre pourvu qu'il n'y ait pas accès simultané).

☞ Services OSEK/VDX pour la gestion des ressources critiques

Le service GetResource

Forme C : Status **GetResource** (ResourceType Resource_ID)

le compte-rendu d'exécution n'est pas utilisé

La tâche appelant ce service rentre en section critique et la ressource nommée lui est attribuée pour usage exclusif.

Le service ReleaseResource

Forme C : Status **ReleaseResource** (ResourceType Resource_ID)

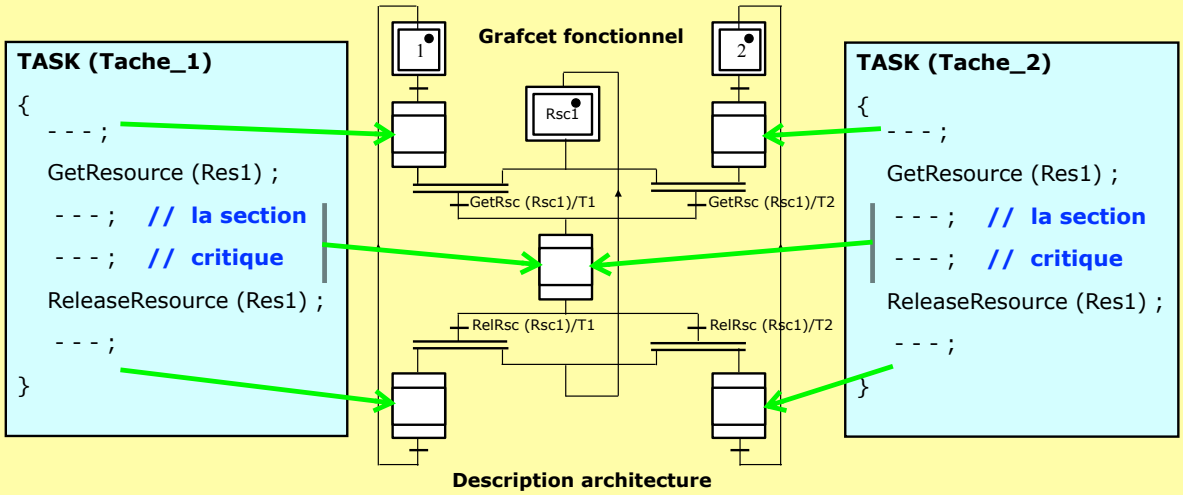
le compte-rendu d'exécution n'est pas utilisé

La tâche appelant ce service libère la section critique et restitue la ressource nommée.

Remarques : **- une ISR est autorisée à appeler ces services.**

- il existe la ressource « Res_scheduler » pour prendre le processeur ⟹ on passe en ordonnancement préemptif.

☞ Usage des services




```
- - - ;
RESOURCE Res1 {
   RESOURCEPROPERTY = STANDARD;
};
- - - ;
```

```
TASK Tache_1 {
   - - - ;
   RESOURCE = Res1 ;
   - - - ;
};
```

```
TASK Tache_2 {
   - - - ;
   RESOURCE = Res1 ;
   - - - ;
};
```

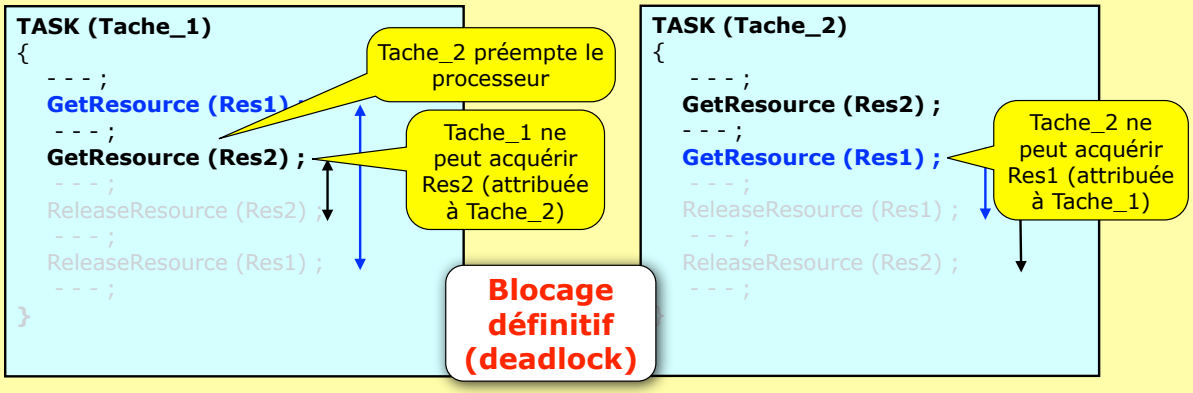
☞ Ressources multiples imbriquées

```
TASK (Tache_1)
{
  - - - ;
  GetResource (Res1) ;
  - - - ;
  GetResource (Res2) ;
  - - - ;
  ReleaseResource (Res2) ;
  - - - ;
  ReleaseResource (Res1) ;
  - - - ;
}
```

```
TASK (Tache_2)
{
  - - - ;
  GetResource (Res1) ;
  - - - ;
  GetResource (Res2) ;
  - - - ;
  ReleaseResource (Res2) ;
  - - - ;
  ReleaseResource (Res1) ;
  - - - ;
}
```

Il faut bien respecter l'imbrication des sections critiques pour éviter l'inter-blocage (deadlock).

☞ Ressources multiples imbriquées



Un exemple peut illustrer le phénomène de « deadlock » (étreinte fatale !) :

T1 : - - - ; Get (R1) ; - - - ; Get (R2) ; - - - ; ---------- etc
T2 : - - - ; Get (R2) ; - - - ; Get (R1) ; - - - ; ---------- etc

La situation ci-dessous est bloquante :

T1 exécute le Get (R1) et est préemptée par T2 avant le Get(R2)
T2 exécute le Get (R2) et ne peut plus prendre R1 qui est déjà prise. Donc blocage définitif.

☞ OSEK/VDX et le protocole PCP

OSEK/VDX utilise une version simplifiée du **protocole PCP** (Priority Ceiling Protocol)

Chaque **ressource** a une **priorité** qui est calculée en fonction des priorités des tâches qui utilisent la ressource (max. des priorités).
Cette valeur de priorité est calculée automatiquement par le compilateur OIL.

Lors de l'invocation du service GetResource la priorité de la tâche appelante est portée à la priorité de la ressource. La Tâche reprend sa priorité initiale lorsqu'elle restitue la ressource.

Ce protocole permet d'éviter :

- ➡ les interblocages
- ➡ l'inversion de priorité (phénomène non commenté ici)

OSEK utilise une version simplifiée de PCP qui est beaucoup plus facile à implémenter.

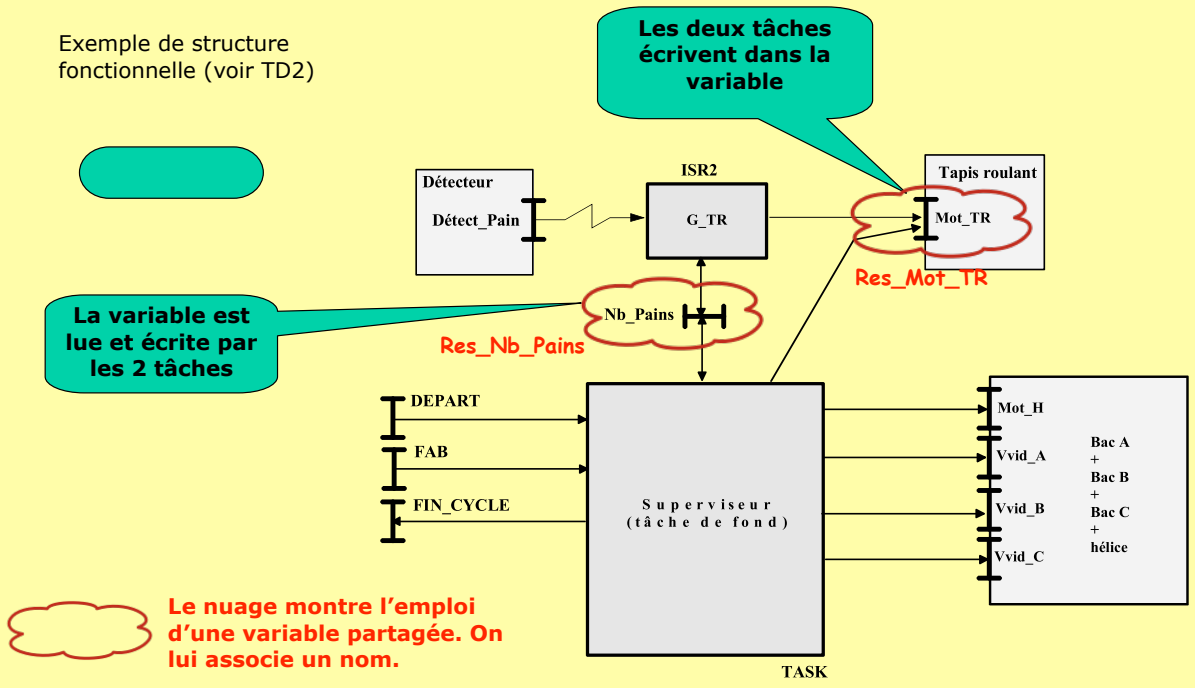
Cependant le fonctionnement du protocole n'est pas expliqué. Il concerne un niveau d'études plus élevé et demande un volume horaire important.

Les ressources étant déclarées dans le langage OIL, ainsi que les tâches qui les utilisent, il est facile de calculer la priorité d'une ressource critique à partir de la priorité des tâches, directement pas le compilateur GOIL.

Pour information, l'inversion de priorité est le phénomène qui peut apparaître avec le protocole PIP (Priority Inheritance Protocol) que l'on retrouve par exemple dans VxWorks. De ce point de vue, OSEK est très en avance.

☞ Communication par variables partagées

Exemple de structure fonctionnelle (voir TD2)



☞ Variables partagées

Une variable partagée doit être protégée (**par principe**) lorsqu'elle est utilisée concurremment par plusieurs entités :

ⓘ utilisation des services d'accès aux ressources critiques

Cependant cela n'est pas toujours utile :

- ➡ lorsque l'accès à la variable est atomique (indivisible)
- ➡ lorsque le protocole d'utilisation rend impossible l'usage simultané

Exemples :

- ➡ Mot_TR est un bit d'un registre, donc accès atomique
- ➡ Nb_Pains est une variable (sur un octet) dont le protocole d'accès garantit le non accès simultané en écriture

Nous sommes ici dans le cas ou toutes les tâches et variables sont locales à un processeur ==> ce sont des variables globales.
Dans le cas d'architectures distribuées il faut utiliser OSEK COM.

☞ Les classes de conformité : pourquoi ?

Créées pour adapter le noyau :

- ➡ aux besoins de l'application
- ➡ aux ressources matérielles disponibles

	tâches basiques seulement	tâches basiques et étendues
pas de mémorisation des activations + **1 tâche / niveau** de priorité	BCC1	ECC1
mémorisation des activations (tâches basiques) + **plusieurs tâches / niveau**	BCC2	ECC2

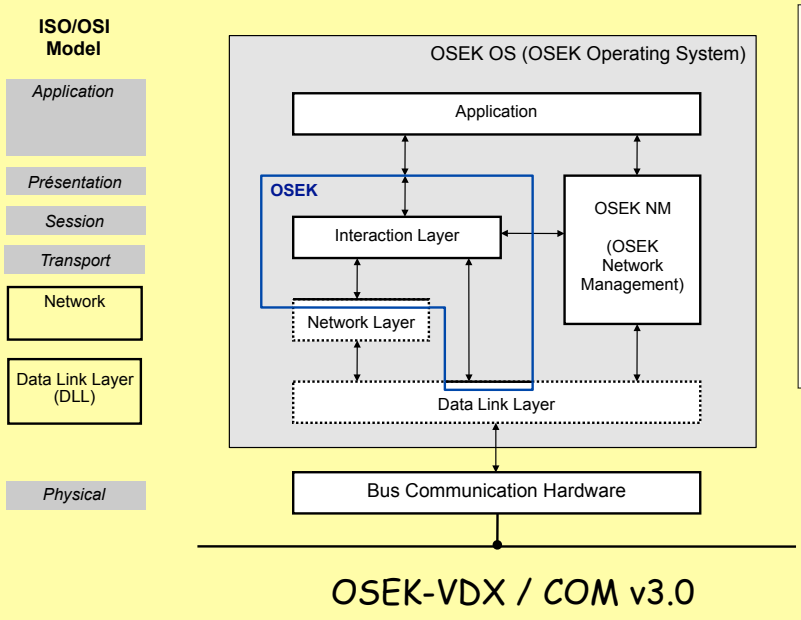
Trampoline est ECC2

☞ Les classes de conformité (autre mode de représentation)

	type de tâches ?	requêtes multiples ?	Nb tâches par niveau ?
BCC1	tâches **basiques**	**1** requête d'activation	**1** priorité / 1 tâche
BCC2	tâches **basiques**	**n** requêtes d'activation (comptage)	**1** priorité / n tâches (liste / sous tableau)
ECC1	tâches **basiques** tâches **étendues**	**1** requête d'activation	**1** priorité / 1 tâche
ECC2	tâches **basiques** tâches **étendues**	**n** requêtes d'activation (tâches basique seulement) (comptage)	**1** priorité / n tâches (liste / sous tableau)

⚠ **L'ordonnancement ne dépend pas de la classe de conformité.**

☞ OSEK/VDX Communication : situation par rapport au modèle OSI



Christophe Marchand, ingénieur PSA, a beaucoup contribué à l'élaboration de la version 3 d'OSEK COM.

Le schéma montre le positionnement de la communication, avec notamment, les interfaces vers les couches Network et DataLink.

Remarque :

OSEK n'a jamais cherché à se positionner par rapport à l'OSI.

OSEK-VDX / COM v3.0

(courtoisie Christophe Marchand (PSA))

☞ L'objet « Message »

- ➡ services basés autour des objets **messages**

  1 message = 1 nom + 1 type de données + des attributs
- ➡ modèle de communication **m : n**
  - ➡ Écriture dans le message : plusieurs écrivains possibles (sur le même site obligatoirement)
  - ➡ Lecture du message : plusieurs lecteurs possibles (un site, sites ≠)
- ➡ des **classes de conformité** pour s'adapter au contexte
  - ➡ CCCA et CCCB : communication **interne** à un site seulement
  - ➡ CCC0 et CCC1 : communication **intra** et **inter-sites**

CAN (Controller Area Network - Principe : Producteur/Consommateur) a influencé les mécanismes utilisés. S'il y a plusieurs écrivains dans le message, ils sont obligatoirement sur le même calculateur (le producteur du message). CAN impose que chaque message possède un identificateur différent. Ceci permet de régler les collisions de manière déterministe. Il ne peut exister deux messages, avec le même identificateur, sur deux calculateurs différents.

Les classes CCCA ou CCCB sont obligatoires. Elles permettent la communication dans un site. Elles différent par le service offert (CCCB permet les files de messages).

Si l'on veut communiquer avec d'autres calculateurs, il faut CCC0 ou CCC1.

☞ Principes

- ➡ Modèle de communication asynchrone
  - ➡ structure « tableau noir » (Unqueued message), variable rafraîchie
  - ➡ structure à file FIFO (Queued message), boîte aux lettre classique
- ➡ pas de suspension de la tâche émettrice pendant la transmission
- ➡ pas de blocage si message non disponible pour la tâche réceptrice

resynchronisation par
- ➡ polling (sur une variable d'état)
- ➡ activation tâche sur fin d'envoi ou réception
- ➡ signalisation d'occurrence sur fin d'envoi ou réception
- ➡ exécution routine Callback
- ➡ alarme sur garde temporelle (émission / réception)

La communication est asynchrone : il n'y a pas de synchronisation entre producteur et consommateurs.

Un message peut prendre deux formes :

- ➡ il est mis à jour par un producteur (éventuellement des producteurs, sur le même site). Les consommateurs lisent toujours la dernière valeur : image du tableau noir ou variable rafraîchie,
- ➡ chaque production est mise en file (structure FIFO, boîte aux lettres classique) et chaque consommation retire un message de la file.

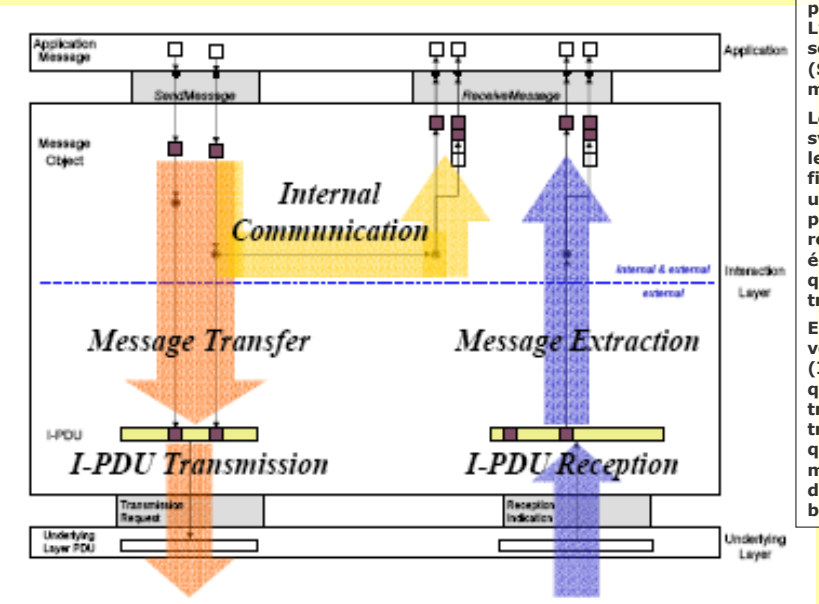
Asynchronisme complet : pas de suspension, que ce soit en émission d'un message ou au contraire en attente.

Bien sûr il faut quand se synchroniser et plusieurs techniques sont offertes :

- ➡ test régulier sur une variable d'état (peu utilisé car mal commode à gérer),
- ➡ activation d'une tâche ou signalisation d'une occurrence d'un événement pour une tâche, en fin d'envoi ou en réception d'un message,
- ➡ exécution d'une routine qui va terminer l'envoi ou l'attente (callback routine).

On peut également protéger temporellement une émission ou une réception via une alarme, qui si elle « saute » fera ce qui est programmé pour l'alarme (activation ou signalisation).

☞ Schéma général simplifié



(Source : documentation OSEK COM)

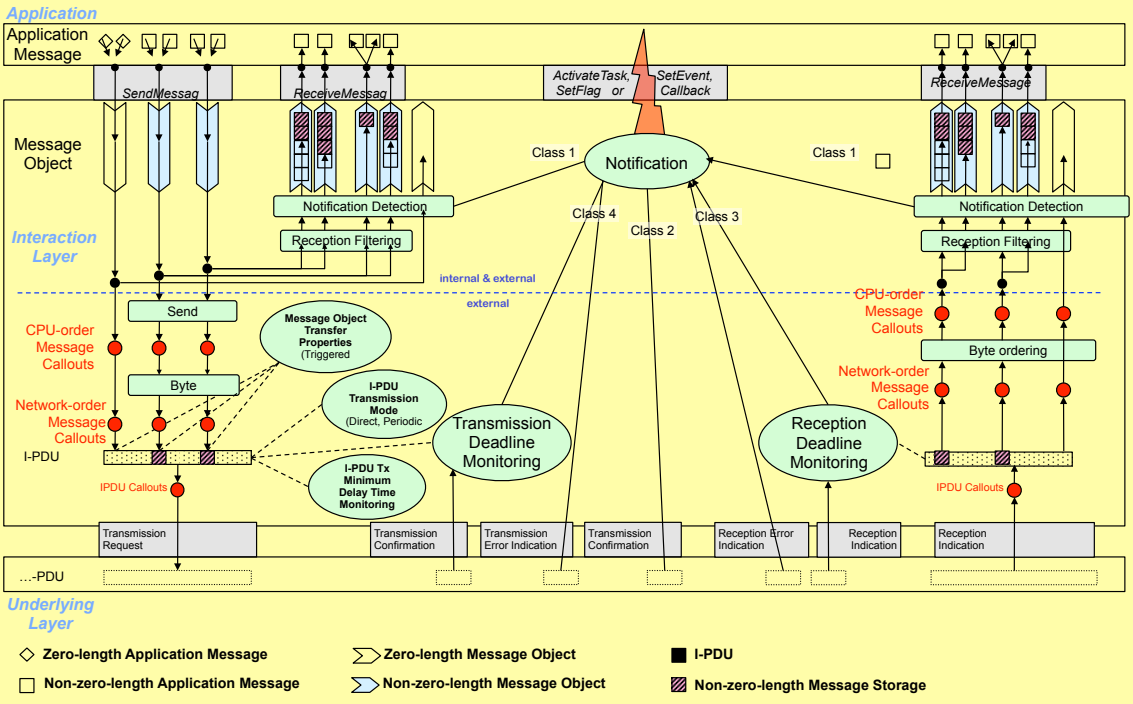
Ce schéma met en évidence les grands chemins suivis par les informations.

Dans l'application, il y a les données (contenu des messages) représentées par les petits carrés blancs. L'application utilise, à l'interface, les services pour envoyer un message (SendMessage) ou recevoir un message (ReceiveMessage).

Les « vrais » messages sont dans le système (Interaction Layer). Ce sont les petits carrés rouges avec ou sans file selon le mode de communication utilisé. Notez les messages qui sont produits (en écriture) et ceux qui sont reçus (en lecture). On voit bien également la communication interne qui ne passe pas par le canal de transmission.

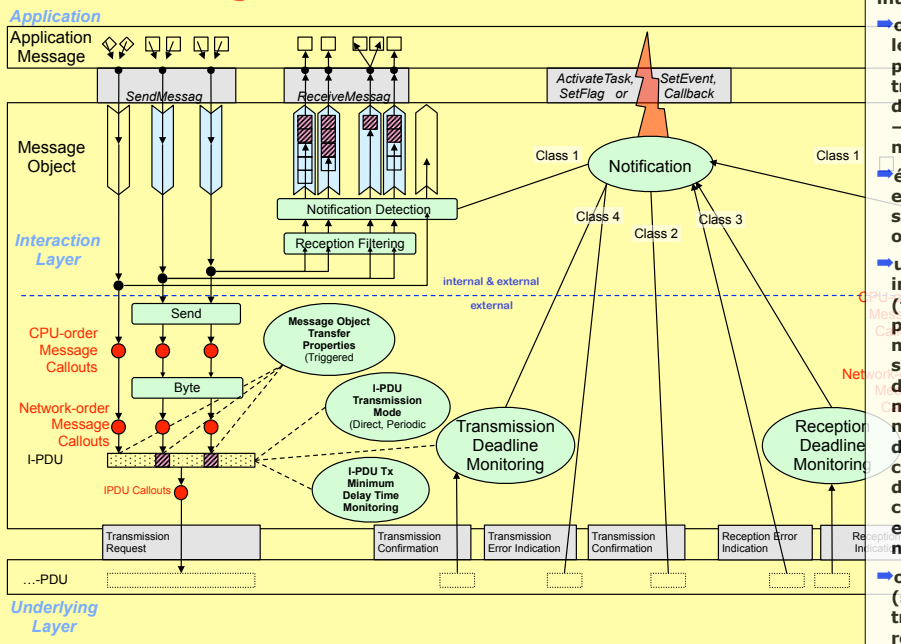
En cas de communication externe on voit sur le dessin les I-PDU (Interaction layer Protocol Data Unit) qui sont les supports pour la transmission et la réception des trames. On voit bien notamment qu'une trame peut contenir plusieurs messages (rationalisation de l'usage du bus : surtout pour des variables binaires).

☞ Schéma général



(courtoisie Christophe Marchand — PSA)

☞ Schéma général



(courtoisie Christophe Marchand — PSA)

Ce schéma très complexe, met en évidence plusieurs éléments intéressants :

- **on peut appliquer des filtres sur les messages (algorithmes qui permet d'autoriser ou non la transmission, la réception et donc, la mise à jour des données — voir doc OSEK-COM si nécessaire)**
- **éventuellement, il faut remettre en place les octets pour que les sites se comprennent (byte ordering)**
- **un message peut être à envoi immédiat après mise à jour (triggered) ou bien attendra la prochaine période d'envoi après mise à jour (pending). Cela suppose que l'émission soit déclarée périodique sur ce message. Les I-PDU ont des modes de transmission variés : direct = modification d'un composant de l'I-PDU → envoi de la trame, Periodic = composition de la trame et envoi périodique, Mixed = un mélange des deux,**
- **on voit le monitoring (surveillance temporelle) des trames en émission et en réception**
- **on voit les notifications (Activate, SetFlag, SetEvent, CallBack) qui peuvent être associées à beaucoup de choses.**